The doBy package – yet another utility package

true

Abstract

The doBy is one of several general utility packages on CRAN. We illustrate two main features of the package. The ability to making groupwise computations and the ability to compute linear estimates, contrasts and least-squares means.

Contents

0.1 Introduction

The `doBy` package [@doby] appeared on CRAN [@CRAN] in 2006 and, much to our surprise, the package is still being used. The package originally grew out of a need to calculate groupwise summary statistics (much in the spirit of `PROC SUMMARY` of the `SAS` system, [@procsummary]). The name comes from `doing` som computations when data is stratified `by` the value of some variables. Today the package contains many different utilities. In this paper we focus 1) on these "doing by" functions, 2) on functions related to linears estimates and contrasts and 3) on some of the miscellaneous functions in the package.

0.2 Functions related to groupwise computations

0.2.1 A working dataset

The CO2 data frame comes from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*. To limit the amount of output we modify names and levels of variables as follows

```
data(CO2)
CO2 <- within(CO2, {
    Treat <- Treatment
    Treatment <- NULL
    levels(Treat) <- c("nchil", "chil")
    levels(Type)  <- c("Que", "Mis")
})
CO2 <- subset(CO2, Plant %in% c("Qn1", "Qc1", "Mn1", "Mc1"))
dim(CO2)
```

```
## [1] 28  5
```

```
head(CO2, 4)
```

```
##   Plant Type conc uptake Treat
## 1   Qn1  Que   95   16.0 nchil
## 2   Qn1  Que  175   30.4 nchil
## 3   Qn1  Que  250   34.8 nchil
## 4   Qn1  Que  350   37.2 nchil
```

0.2.2 The summaryBy function

The summaryBy function is used for calculating quantities like *the mean and variance of numerical variables x and y for each combination of two factors A and $B$$*. Notice: A functionality similar to summaryBy is provided by aggregate from base R, but summaryBy offers additional features.

```
myfun1 <- function(x){c(m=mean(x), s=sd(x))}
summaryBy(cbind(conc, uptake, lu=log(uptake)) ~ Plant, data=CO2, FUN=myfun1)
```

```
##   Plant conc.m conc.s uptake.m uptake.s  lu.m   lu.s
## 1   Qn1    435  317.7    33.23    8.215 3.467 0.3189
## 2   Qc1    435  317.7    29.97    8.335 3.356 0.3446
## 3   Mn1    435  317.7    26.40    8.694 3.209 0.4234
## 4   Mc1    435  317.7    18.00    4.119 2.864 0.2622
```

```
## same as
## aggregate(cbind(conc, uptake, log(uptake)) ~ Plant, data=CO2, FUN=myfun1)
```

The convention is that variables that do not appear in the dataframe (e.g. log(uptake)) must be named (here as lu).

It is generally the the case the “By”-functions that a two sided formula like can be written in two ways:

```
cbind(x, y) ~ A + B
list(c("x", "y"), c("A", "B"))
```

The list form is if summaryBy is invoked programatically (this is a feature of summaryBy and it does not work with aggregate):

```
summaryBy(list(c("conc", "uptake", "lu=log(uptake)"), "Plant"), data=CO2, FUN=myfun1)
```

Various convenient abbreviations are available, e.g. the following, where left hand side dot refers to "all numeric variables" while the right hand side dot refers to "all factor variables". Writing `1` on the right hand side leads to computing over the entire dataset:

```
summaryBy(.~., data=CO2, FUN=myfun1)
```

```
##   Plant Type Treat conc.m conc.s uptake.m uptake.s
## 1   Qn1  Que nchil    435  317.7    33.23    8.215
## 2   Qc1  Que  chil    435  317.7    29.97    8.335
## 3   Mn1  Mis nchil    435  317.7    26.40    8.694
## 4   Mc1  Mis  chil    435  317.7    18.00    4.119
```

```
summaryBy(.~1, data=CO2, FUN=myfun1)
```

```
##   conc.m conc.s uptake.m uptake.s
## 1    435  299.6     26.9    9.189
```

The shorthand notation is

```
summaryBy(list(c("."), c(".")), data=CO2, FUN=myfun1)
summaryBy(list(c("."), c("1")), data=CO2, FUN=myfun1)
```

0.2.3 The `orderBy` function

Ordering (or sorting) a data frame is possible with the `orderBy` function. Suppose we want to order the rows of the the `CO2` data by increasing values of `conc` and decreasing value of `uptake` (within `code`):

```
x1 <- orderBy(~ conc - uptake, data=CO2)
head(x1)
```

```
##    Plant Type conc uptake Treat
## 1    Qn1  Que   95   16.0 nchil
## 22   Qc1  Que   95   14.2  chil
## 43   Mn1  Mis   95   10.6 nchil
## 64   Mc1  Mis   95   10.5  chil
## 2    Qn1  Que  175   30.4 nchil
## 23   Qc1  Que  175   24.1  chil
```

It is generally the the case the "By"-functions that a right hand sided formula can be written in two ways:

```
~ A + B
c("A", "B")
```

An equivalent form is therefore

```
orderBy(c("conc", "-uptake"), data=CO2)
```

0.2.4 The `splitBy` function

Suppose we want to split `CO2` into a list of dataframes:

```
x1 <- splitBy(~ Plant + Type, data=CO2)
x1
```

```
##   listentry Plant Type
## 1   Qn1|Que   Qn1  Que
## 2   Qc1|Que   Qc1  Que
## 3   Mn1|Mis   Mn1  Mis
## 4   Mc1|Mis   Mc1  Mis
```

The result is a list (with a few additional attributes):

```
lapply(x1, head, 2)
```

```
## $`Qn1|Que`
##   Plant Type conc uptake Treat
## 1   Qn1  Que   95   16.0 nchil
## 2   Qn1  Que  175   30.4 nchil
##
## $`Qc1|Que`
##    Plant Type conc uptake Treat
## 22   Qc1  Que   95   14.2  chil
## 23   Qc1  Que  175   24.1  chil
##
## $`Mn1|Mis`
##    Plant Type conc uptake Treat
## 43   Mn1  Mis   95   10.6 nchil
## 44   Mn1  Mis  175   19.2 nchil
##
## $`Mc1|Mis`
##    Plant Type conc uptake Treat
## 64   Mc1  Mis   95   10.5  chil
## 65   Mc1  Mis  175   14.9  chil
```

0.2.5 The `subsetBy` function

Suppose we want to select those rows within each treatment for which the uptake is larger than 75% quantile of uptake (within the treatment). This is achieved by:

```
x2 <- subsetBy(~Treat, subset=uptake > quantile(uptake, prob=0.75), data=CO2)
head(x2, 4)
```

```
##          Plant Type conc uptake Treat
## nchil.4    Qn1  Que  350   37.2 nchil
## nchil.6    Qn1  Que  675   39.2 nchil
## nchil.7    Qn1  Que 1000   39.7 nchil
## nchil.49   Mn1  Mis 1000   35.5 nchil
```

0.2.6 The `transformBy` function

The `transformBy` function is analogous to the `transform` function except that it works within groups. For example:

```
x3 <- transformBy(~Treat, data=CO2,
                  minU=min(uptake), maxU=max(uptake),
                  range=diff(range(uptake)))
head(x3, 4)
```

```
##         Plant Type conc uptake Treat minU maxU range
## nchil.1   Qn1  Que   95   16.0 nchil 10.6 39.7  29.1
## nchil.2   Qn1  Que  175   30.4 nchil 10.6 39.7  29.1
## nchil.3   Qn1  Que  250   34.8 nchil 10.6 39.7  29.1
## nchil.4   Qn1  Que  350   37.2 nchil 10.6 39.7  29.1
```

0.2.7 The `lmBy` function

The `lmBy` function allows for fitting linear models to different strata of data:

```
m <- lmBy(uptake ~ conc | Treat, data=CO2)
coef(m)
```

```
##       (Intercept)    conc
## nchil       20.82 0.02067
## chil        17.02 0.01602
```

The result is a list with a few additional attributes and the list can be processed further as e.g.

```
lapply(m, function(z) coef(summary(z)))
```

```
## $nchil
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept) 20.82342   3.092430   6.734 2.092e-05
## conc         0.02067   0.005889   3.510 4.304e-03
##
## $chil
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept) 17.01814   3.668315   4.639 0.0005709
## conc         0.01602   0.006986   2.293 0.0407168
```

0.3 Functions related linear estimates and contrasts

A linear function of a p–dimensional parameter vector β has the form

$$C = L\beta$$

where L is a $q \times p$ matrix which we call the `Linear Estimate Matrix` or simply `LE-matrix`. The corresponding linear estimate is $\hat{C} = L\hat{\beta}$. A linear hypothesis has the form $H_0 : L\beta = m$ for some q dimensional vector m.

0.3.1 A working dataset

The response is the length of odontoblasts cells (cells responsible for tooth growth) in 60 guinea pigs. Each animal received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods, (orange juice (coded as OJ) or ascorbic acid (a form of vitamin C and (coded as VC)).

```
ToothGrowth$dose <- factor(ToothGrowth$dose)
head(ToothGrowth, 4)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
```

The interaction plot indicates some interaction between `dose` and `supp`.

This is also supported by a formal test:

```
tooth1 <- lm(len ~ dose + supp, data=ToothGrowth)
tooth2 <- lm(len ~ dose * supp, data=ToothGrowth)
anova(tooth1, tooth2)
```

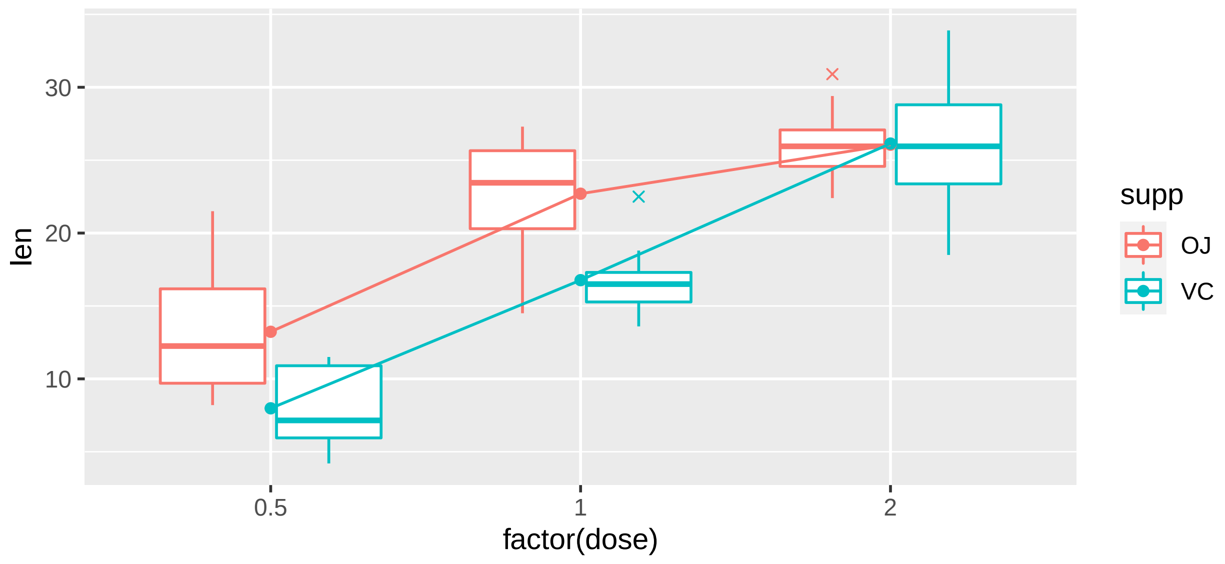

Figure 1: Interaction plot for the ToothGrowth data. The average `len` for each group is a dot. Boxplot outliers are crosses.

```
## Analysis of Variance Table
##
## Model 1: len ~ dose + supp
## Model 2: len ~ dose * supp
##   Res.Df RSS Df Sum of Sq    F Pr(>F)
## 1     56 820
## 2     54 712  2       108 4.11  0.022 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

0.3.2 Computing linear estimates

For now, we focus on the additive model. Consider computing the estimated length for each dose of orange juice (OJ): One option: Construct the LE–matrix L directly and then invoke `linest`:

```
L <- matrix(c(1, 0, 0, 0,
              1, 1, 0, 0,
              1, 0, 1, 0), nrow=3, byrow=T)
c1 <- linest(tooth1, L)
c1
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   12.455     0.988    12.603 56.000       0
## [2,]   21.585     0.988    21.841 56.000       0
## [3,]   27.950     0.988    28.281 56.000       0
```

We can do:

```
summary(c1)
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   12.455     0.988    12.603 56.000       0
## [2,]   21.585     0.988    21.841 56.000       0
## [3,]   27.950     0.988    28.281 56.000       0
##
## Grid:
## NULL
##
## L:
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    1    1    0    0
## [3,]    1    0    1    0
```

```
coef(c1)
```

```
##   estimate std.error statistic df   p.value
## 1    12.45    0.9883     12.60 56 5.490e-18
## 2    21.58    0.9883     21.84 56 4.461e-29
## 3    27.95    0.9883     28.28 56 7.627e-35
```

```
confint(c1)
```

```
##   0.025 0.975
## 1 10.48 14.43
## 2 19.61 23.56
## 3 25.97 29.93
```

The matrix L can be generated as follows:

```
L <- LE_matrix(tooth1, effect="dose", at=list(supp="OJ"))
L
```

```
##      (Intercept) dose1 dose2 suppVC
## [1,]           1     0     0      0
## [2,]           1     1     0      0
## [3,]           1     0     1      0
```

There are various alternatives:

```
c1 <- esticon(tooth1, L)
c1
```

```
##      estimate std.error statistic p.value  beta0 df
## [1,]   12.455     0.988    12.603   0.000  0.000 56
## [2,]   21.585     0.988    21.841   0.000  0.000 56
## [3,]   27.950     0.988    28.281   0.000  0.000 56
```

Notice: `esticon` has been in the `doBy` package for many years; `linest` is a newer addition. Yet another alternative in this case is to generate a new data frame and then invoke predict (but this approach is not generally applicable, see later):

```
nd <- data.frame(dose=c('0.5', '1', '2'), supp='OJ')
nd
```

```
##   dose supp
## 1  0.5   OJ
## 2    1   OJ
## 3    2   OJ
```

```
predict(tooth1, newdata=nd)
```

```
##     1     2     3
## 12.45 21.58 27.95
```

0.3.3 Least-squares means (LS–means)

A related question could be: What is the estimated length for each dose if we ignore the source of vitamin C (i.e. whether it is OJ or VC). One approach would be to fit a model in which source does not appear:

```
tooth0 <- update(tooth1, . ~ . - supp)
L0 <- LE_matrix(tooth0, effect="dose")
L0
```

```
##      (Intercept) dose1 dose2
## [1,]           1     0     0
## [2,]           1     1     0
## [3,]           1     0     1
```

```
linest(tooth0, L=L0)
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   10.605     0.949    11.180 57.000       0
## [2,]   19.735     0.949    20.805 57.000       0
## [3,]   26.100     0.949    27.515 57.000       0
```

An alternative would be to stick to the original model but compute the estimate for an "average vitamin C source". That would correspond to giving weight 1/2 to each of the two vitamin C source parameters. However, as one of the parameters is already set to zero to obtain identifiability, we obtain the LE–matrix L as

```
L1 <- matrix(c(1, 0, 0, 0.5,
               1, 1, 0, 0.5,
               1, 0, 1, 0.5), nrow=3, byrow=T)
linest(tooth1, L=L1)
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   10.605     0.856    12.391 56.000       0
## [2,]   19.735     0.856    23.058 56.000       0
## [3,]   26.100     0.856    30.495 56.000       0
```

Such a particular linear estimate is sometimes called a least-squares mean or an LSmean or a marginal mean. Notice that the parameter estimates under the two approaches are identical. This is is because data is balanced: There are 10 observations per supplementation type. Had data not been balanced, the estimates would in general have been different.

Notice: One may generate L automatically with

```
L1 <- LE_matrix(tooth1, effect="dose")
L1
```

```
##      (Intercept) dose1 dose2 suppVC
## [1,]           1     0     0    0.5
## [2,]           1     1     0    0.5
## [3,]           1     0     1    0.5
```

Notice: One may obtain the LSmean directly as:

```
LSmeans(tooth1, effect="dose")
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   10.605     0.856    12.391 56.000       0
## [2,]   19.735     0.856    23.058 56.000       0
## [3,]   26.100     0.856    30.495 56.000       0
```

which is the same as

```
L <- LE_matrix(tooth1, effect="dose")
le <- linest(tooth1, L=L)
coef(le)
```

```
##   estimate std.error statistic df   p.value
## 1    10.60    0.8559     12.39 56 1.109e-17
## 2    19.73    0.8559     23.06 56 2.885e-30
## 3    26.10    0.8559     30.50 56 1.444e-36
```

0.3.4 Interaction model

For a model with interactions, the LSmeans are

```
LSmeans(tooth2, effect="dose")
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   10.605     0.812    13.060 54.000       0
## [2,]   19.735     0.812    24.304 54.000       0
## [3,]   26.100     0.812    32.143 54.000       0
```

In this case, the LE–matrix is

```
L <- LE_matrix(tooth2, effect="dose")
t(L)
```

```
##                [,1] [,2] [,3]
## (Intercept)     1.0  1.0  1.0
## dose1           0.0  1.0  0.0
## dose2           0.0  0.0  1.0
## suppVC          0.5  0.5  0.5
## dose1:suppVC    0.0  0.5  0.0
## dose2:suppVC    0.0  0.0  0.5
```

0.3.5 Using (transformed) covariates

Below, the covariate `conc` is fixed at the average value:

```
co2.lm1 <- lm(uptake ~ conc + Type + Treat, data=CO2)
LSmeans(co2.lm1, effect="Treat")
```

```
## Coefficients:
##      estimate std.error statistic    df p.value
## [1,]    29.81      1.35     22.16 24.00       0
## [2,]    23.99      1.35     17.83 24.00       0
```

If we use `log(conc)` instead we will get an error when calculating LS–means because `log(conc)` is not a variable in the dataframe. Instead one can do:

```r
co2.lm2 <- lm(uptake ~ log.conc + Type + Treat,
              data=transform(CO2, log.conc=log(conc)))
LSmeans(co2.lm2, effect="Treat")
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   29.814     0.934    31.938 24.000       0
## [2,]   23.986     0.934    25.694 24.000       0
```

This also highlights what is computed: The average of the log of `conc`; not the log of the average of `conc`. In a similar spirit consider

```r
co2.lm3 <- lm(uptake ~ conc + I(conc^2) + Type + Treat, data=CO2)
LSmeans(co2.lm3, effect="Treat")
```

```
## Coefficients:
##      estimate std.error statistic    df p.value
## [1,]    33.24      1.29     25.81 23.00       0
## [2,]    27.41      1.29     21.29 23.00       0
```

Above `I(conc^2)` is the average of the squared values of `conc`; not the square of the average of `conc`, cfr. the following.

```r
co2.lm4 <- lm(uptake ~ conc + conc2 + Type + Treat, data=
               transform(CO2, conc2=conc^2))
LSmeans(co2.lm4, effect="Treat")
```

```
## Coefficients:
##      estimate std.error statistic    df p.value
## [1,]    29.81      1.02     29.27 23.00       0
## [2,]    23.99      1.02     23.55 23.00       0
```

If we want to evaluate the LS–means at `conc=10` then we can do:

```r
LSmeans(co2.lm4, effect="Treat", at=list(conc=10, conc2=100))
```

```
## Coefficients:
##      estimate std.error statistic    df p.value
## [1,]    14.67      2.23      6.58 23.00       0
## [2,]     8.84      2.23      3.96 23.00       0
```

0.4 Alternative models

0.4.1 Generalized linear models

We can calculate LS–means for e.g. a Poisson or a gamma model. Default is that the calculation is calculated on the scale of the linear predictor. However, if we think of LS–means as a prediction on the linear scale one may argue that it can also make sense to transform this prediction to the response scale:

```r
tooth.gam <- glm(len ~ dose + supp, family=Gamma, data=ToothGrowth)
LSmeans(tooth.gam, effect="dose", type="link")
```

```
## Coefficients:
##      estimate std.error statistic p.value
## [1,]  0.09453   0.00579  16.33340       0
## [2,]  0.05111   0.00312  16.39673       0
## [3,]  0.03889   0.00238  16.36460       0
```

```
LSmeans(tooth.gam, effect="dose", type="response")
```

```
## Coefficients:
##      estimate std.error statistic p.value
## [1,]  0.09453   0.00579  16.33340       0
## [2,]  0.05111   0.00312  16.39673       0
## [3,]  0.03889   0.00238  16.36460       0
```

0.4.2 Linear mixed effects model

For the sake of illustration we treat `supp` as a random effect:

```
library(lme4)
```

```
## Loading required package: Matrix
```

```
tooth.mix <- lmer( len ~ dose  + (1|supp), data=ToothGrowth)
LSmeans(tooth1, effect="dose")
```

```
## Coefficients:
##      estimate std.error statistic     df p.value
## [1,]   10.605     0.856    12.391 56.000       0
## [2,]   19.735     0.856    23.058 56.000       0
## [3,]   26.100     0.856    30.495 56.000       0
```

```
LSmeans(tooth.mix, effect="dose")
```

```
## Coefficients:
##      estimate std.error statistic    df p.value
## [1,]    10.61      1.98      5.36  1.31    0.08
## [2,]    19.74      1.98      9.98  1.31    0.03
## [3,]    26.10      1.98     13.20  1.31    0.02
```

Notice here that the estimates themselves identical to those of a linear model (that is not generally the case, but it is so here because data is balanced). In general the estimates are will be very similar but the standard errors are much larger under the mixed model. This comes from that there that `supp` is treated as a random effect.

```
VarCorr(tooth.mix)
```

```
##  Groups   Name        Std.Dev.
##  supp     (Intercept) 2.52
##  Residual             3.83
```

Notice that the degrees of freedom by default are adjusted using a Kenward–Roger approximation (provided that `pbkrtest` is installed). Unadjusted degrees of freedom are obtained by setting `adjust.df=FALSE`.

```
LSmeans(tooth.mix, effect="dose", adjust.df=FALSE)
```

```
## Coefficients:
##      estimate std.error statistic    df p.value
## [1,]    10.61      1.98      5.36 55.00       0
## [2,]    19.74      1.98      9.98 55.00       0
## [3,]    26.10      1.98     13.20 55.00       0
```

0.4.3 Generalized estimating equations

Lastly, for gee-type "models" we get

```
library(geepack)
tooth.gee <- geeglm(len ~ dose, id=supp, family=Gamma, data=ToothGrowth)
LSmeans(tooth.gee, effect="dose")
```

```
## Coefficients:
##      estimate std.error statistic p.value
## [1,] 9.43e-02  1.65e-02  5.71e+00       0
## [2,] 5.07e-02  5.38e-03  9.41e+00       0
## [3,] 3.83e-02  4.15e-05  9.23e+02       0
```

```
LSmeans(tooth.gee, effect="dose", type="response")
```

```
## Coefficients:
##      estimate std.error statistic p.value
## [1,] 9.43e-02  1.65e-02  5.71e+00       0
## [2,] 5.07e-02  5.38e-03  9.41e+00       0
## [3,] 3.83e-02  4.15e-05  9.23e+02       0
```

0.5 Acknowledgements

Credit is due to Dennis Chabot, Gabor Grothendieck, Paul Murrell, Jim Robison-Cox and Erik Jørgensen for reporting various bugs and making various suggestions to the functionality in the doBy package.